Chapter 15

Some Problems with Web Programs

Web programs are notoriously buggy. For instance, consider the following interaction with a popular commercial travel Web site.

1. Choose the option to search for a hotel room, and enter the corresponding information.

2. Suppose the Web site response with two hotels, options A and B.

3. Using your browser’s interactive facilities, open the link to hotel A in a separate window.

4. Suppose you find A reasonable, but are curious about the details of B. You therefore return to the window listing all the hotels and open the details for B in a separate window.

5. Having scanned the details of B, you find A a more attractive option. Since the window for A is still on-screen, you switch to it and click the reservation link.

6. The travel site makes your reservation at hotel B.

If an error like this were isolated to a single Web page, or even to a single site, we can put it down to programmer error. But when the same error occurs on numerous sites, it forces us to systematically investigate their cause and to more carefully consider the design and implementation of Web programs.

Before we investigate the problem in general, it helps to understand its breadth. The following is an uncontroversial property that we would expect of a travel reservation site:

The user should receive a reservation at the hotel that was displayed on the page he submitted.

Can we generalize this? That is, should a user receive information based strictly on the information displayed on the page on which the user clicked a button?

This appears reasonable at first blush. For instance, software that satisfies it avoids the embarrassing problem that we saw above, where abandoned investigations can nonetheless affect the program’s output. But consider an airline reservation system, where a user must choose between multiple flights for each of the outbound and return legs. Depending on what information is presented on each page, it may not: in a poor Web design, the page that displays the return leg choices may not display the chosen outward leg. But
even if the site were designed to avoid this problem, there are sites where we do want “interference” from other explored options.

Consider an on-line bookstore. Conduct the same sequence of interactions as above, except with books instead of hotels. Upon examining choice B, suppose you clicked to add it to your “shopping cart”. Now when you go to the page for book A and add it, too, to your shopping cart, what do you expect to find in it? Certainly, the bookseller hopes you have both A and B in the cart (since, after all, they are in the business of selling as many books as possible). This is a clear violation of the property we elucidated above.

The problem is compounded by the number of interaction operations supported by modern Web browsers. In addition to opening Web pages in new windows, browsers offer the ability to clone the currently-visible page, to go back to a previous page, to go forward to a page (from which the user had previously gone back), to create bookmarks, and so on. The number of interactive operations appears to grow with each browser release. Worse, most of these operations are silent: the browser does not notify the Web application that they have been executed, so the application must reconstruct events based on the submissions it receives.

Many of the problems with Web programs trace back to their structure. The Web’s architecture dictates that every time a Web program sends an Web page to a user, it is forced to terminate; this is because the Web implements a stateless protocol. If and when the user chooses to resume the computation (by clicking on a link or button), some other program must resume the computation. This forces a rather perverse program structure on the programmer. We will now study the implications of this structure in some detail.
Stateful and Stateless Protocols

Suppose a client-server computation relies on performing multiple interactions. In a stateful protocol, the server maintains some state information recording its context in the dialog. A well-known example of a stateful protocol is FTP, the Internet file-transfer protocol. In an FTP session, the user can enter multiple commands, and the interpretation of each command is relative to the history of past commands. That is, two invocations of `ls` (to list files in a directory) will produce different answers if the user has invoked `cd` (to change the directory) betwixt. (The context information here is the current directory.)

In contrast to many traditional Internet protocols, the Web implements a stateless protocol, meaning it does not retain any record of prior communication. As a result, in principle the Web application is responsible for completely restoring the state of the computation on each interaction. By analogy, suppose you were executing an editor such as Emacs within an SSH session (which is also stateful: this state is lost when the connection dies). In a stateless SSH, after every unit of output the connection would close. When the user entered another keystroke, the communication would have to carry with it the entire state of running applications (indeed, the entire history, to enable Undo operations), the server would have to invoke Emacs afresh, run all the commands entered so far and, having restored the application to its past state... enter the new keystroke.

Put this way, a stateless protocol seems quite ludicrous. So why would anyone employ a stateless protocol? They confer the advantage that the server has to tolerate far lower loads. If a server can maintain only 1,000 connections at any instant, in a stateful protocol (that keeps connections open until the transaction terminates) the server would not be able to service more than 1,000 users at a time. Worse, it would need a policy for determining when to terminate connections that appear to no longer be active. A stateless protocol avoids this; the server can serve many more clients in rapid order, and can ignore clients who are not interested in completing a computation. It pays the price of transmitting enough data to resume the computation.

Using a stateless protocol for very fine-grained communication (as in a text editor) is obviously a bad idea. Most Web applications are therefore instead designed to communicate at a coarse granularity. That said, stateful protocols are easier to program because the developer is not responsible for setup and breakdown of the state at each interaction. Thus, APIs such as Java servlets, or especially the implementation of `web-read/k` we have posited above, relieve some of this burden by providing a partially stateful interface to the Web developer. The developer must, however, pay the price of determining how to manage resources. If a Web user does not explicitly “log out” or otherwise signal premature termination, when can the server reap the corresponding session object?
Chapter 16

The Structure of Web Programs

Suppose we are trying to implement the following simple Web program. The program presents the user with a prompt for a number. Given an input, it presents a prompt for a second number. Given a second input, it displays the sum of the two numbers in a Web page:

```
(web-display
 (+ (web-read "First number: ")
    (web-read "Second number: ")))
```

While this is an extremely simple application, it is sufficient for demonstrating many concepts. Furthermore, it is a microcosm of a more serious Web application such as one that consumes name, address and credit card information in multiple stages—

```
(web-display
 (purchase-item (web-read "Name: ")
                (web-read "Credit Card Number: ")))
```

—or one that offers multiple flight choices for the outward and return legs of a flight and makes a reservation given the two choices:

```
(web-display
 (make-reservation (web-read "Select outward flight: ")
                   (web-read "Select return flight: ")))  
```

Even this “addition server” is difficult to implement:

1. The Web developer must turn this application into three programs:

   (a) The first program displays the first form.

   (b) The second program consumes the form values from the first form, and generates the second form.

   
```
1We are assuming the existence of some simple primitives that mask the necessary but, for now, irrelevant complexity of generating HTML forms, and so on.
```
(c) The third program consumes the form values from the second form, computes the output, and generates the result.

2. Because the value entered in the first form is needed by the third program to compute its output, this value must somehow be transmitted between from the first program to the third. This is typically done by using the hidden field mechanism of HTML.

3. Suppose, instead of using a hidden field, the application developer used a Java Servlet session object, or a database field, to store the first number. (Application developers are often pushed to do this because that is the feature most conveniently supported by the language and API they are employing.) Then, if the developer were to exploratorily open multiple windows, as we discussed in Section 5, the application can compute the wrong answer.

In particular, the programs we have written above, which would appear to be perfectly normal programs to run on a display console, cannot run on the Web: the moment web-read dispatches its Web form to the user, the Web protocol forces the computation to terminate, taking with it all memory of what had to happen next, i.e., the pending computation.

Where is this pending computation specified? The system resumes execution at the URL specified in the “action” field of the generated form. The developer is therefore responsible for making sure that the application that resides at that URL is capable of resuming the computation in its entirety. We have entirely neglected this problem by assuming the existence of a web-read procedure, but in fact the entire problem is that we cannot implement it without a more explicit handle on the pending computation.

16.1 Explicating the Pending Computation

For our motivating example, what is the pending computation at the point of the first interaction? In words, it is to consume the result from the form (the first number), generate a form for the second number, add them, then display their result. Since natural language is unwieldy, we would benefit from writing this pending computation in code instead:

\[
\text{(web-display)} \\
\qquad (+ \bullet) \\
\qquad \text{(web-read "Second number: ")})
\]

where we use \( \bullet \) to represent the result from the user’s form submission. What is \( \bullet \), exactly? It appears to be an invented notation that we must then explain formally. Instead, we can treat it as an identifier, binding it in the traditional way:

\[
\text{lambda (\bullet)} \\
\qquad \text{(web-display)} \\
\qquad \qquad (+ \bullet) \\
\qquad \qquad \text{(web-read "Second number: ")})
\]

This procedure, then, represents the computation pending at the point of the first interaction. Apply this procedure to the result of that interaction, and it should successfully resume the computation. Similarly, the pending computation at the point of the second interaction is
16.2. A BETTER SERVER PRIMITIVE

Suppose, therefore, that we had a modified version of \texttt{web-read} that we’ll call \texttt{web-read/k}. This new procedure takes two arguments. The first is a string that it converts into a form, as before. The second is a procedure of one argument representing the pending computation, which we’ll henceforth call the \textit{receiver}.

Every time \texttt{web-read/k} is invoked, it creates a fresh entry in a hash table. It stores the receiver in this entry, and generates a form action URL that contains the hash table key for this procedure. The hash table is kept in memory by the Web server (which, we’ll assume, doesn’t terminate). \texttt{web-read/k} generates a page and then terminates the Web application’s execution, in conformance with the Web protocol.

This generated page is shown in Figure [16.1] The image shows, in outline, the Web page generated by invoking \texttt{web-read/k} with the first argument reading "First". This string becomes the prompt. Next to the prompt is a text box where the user can provide input. The action field of the HTML form has a reference to the hash table key of the corresponding fresh entry (in this instance, \texttt{k2592}).

When the user submits a response, the server invokes the application named \texttt{launch}. This application does two things. First, it uses the key associated with the \texttt{id} argument to obtain a receiver closure from the hash table. Second, it extracts the input typed by the user in the text box. The receiver is then applied to this extracted value. This resumes the computation.

Assuming the existence of such a primitive, we might try to rewrite our running application as

\begin{verbatim}
(lambda (k)
  (web-display
    (+ k
      \texttt{"First number: "}))
\end{verbatim}

where \(k\) is the user’s response to the first prompt, which is presumably in the closure of this procedure.
(lambda (•)
   •))
(web-read "Second number: ")

but this won’t work at all! Recall that at every Web interaction, the Web application entirely terminates. That means, any computation that has not been included in the receiver is lost forever. As a consequence, when this application resumes, the only “remembered” computation is that in the receiver, which is just the identity function: the second Web input, as well as the ensuing computation and the display of the result, have all been irretrievably lost.

In other words, any computation that isn’t explicitly mentioned in the receiver simply never gets performed, because of the program’s termination after each interaction. This forces us to move all pending computation into the receiver. Here’s what we might try:

(web-read/k "First number: 
  (lambda (•)
    (web-display
      (+ •
        (web-read "Second number: ")))))

This, however, is subject to the same analysis: it still uses the hypothetical web-read procedure, which we’ve conceded we don’t quite know how to implement. We must, therefore, instead employ web-read/k again, as follows:

(web-read/k "First number: 
  (lambda (•)
    (web-read/k "Second number: 
      (lambda (•)
        (web-display
          (+ • •))))))

Oh, not quite: we want to add the first number to the second, not just compute twice the second number. Therefore:

(web-read/k "First number: 
  (lambda (•₁)
    (web-read/k "Second number: 
      (lambda (•₂)
        (web-display
          (+ •₁ •₂))))))

Now, when the program finally generates the sum, it can safely halt without having registered any receivers, because there aren’t any computations left to perform. Relative to the original source program, however, the structure of this application is considerably more intricate.

**Exercise 16.2.1** To be entirely pedantic, there is one thing left to do, which is to explicitly halt the program. Extend the program to do this, then transform it to correctly employ web-read/k.
16.3 Testing Web Transformations

One of the subtle problems with transforming interactive programs for the Web is that they are difficult to test. This difficulty has at least two facets. First, the use of HTML makes programs unwieldy, so we would rather defer its use until the end, but without it we cannot interact with a Web browser. Second, testing a program at the console can be misleading: a computation may not have been properly moved into a receiver but, because Scheme programs do not terminate after every interaction, we would never notice this problem until we ran the program on the Web.

Fortunately, it is easy to simulate the Web’s behavior at the console with the following code. The following implementation of \texttt{web-read/k} stores the receiver and prompt in a box, and terminates the program’s execution using \texttt{error}:

\input{16.3-testing-web-transformations}

The procedure \texttt{resume} uses the values in these boxes to resume the computation:

\input{16.3-testing-web-transformations}

We can therefore test a program such as the addition application as follows:

Welcome to DrScheme, version 208p1.
Language: PLAI - Advanced Student.
web-read/k: run (resume) to enter number and simulate clicking Submit
>

This means the program has arrived at \texttt{web-read/k} for the first time. We run

\input{16.3-testing-web-transformations}

which prompts us for the first input. Providing an input results in the same terminating “error” message, corresponding to the next interaction point. Running \texttt{(resume)} prompts for the second input. When we provide the second, we see the sum of the two numbers printed to the console.
16.4 Executing Programs on a Traditional Server

Suppose we must run our Web application on a traditional Web server, which does not provide support for
the hash table used by \texttt{web-read/k}. This doesn’t mean we must waste the effort we expended transforming
the program: that effort was a direct consequence of the Web’s protocol, which the traditional server also
obeys (even more slavishly!).

What’s the problem with executing this program on a traditional server?

\[
(\texttt{web-read/k "First number: "} \ \\
\quad \ \texttt{(lambda (\_1)} \ \\
\quad \quad \texttt{(web-read/k "Second number: "} \ \\
\quad \quad \quad \texttt{(lambda (\_2)} \ \\
\quad \quad \quad \quad \texttt{(web-display}} \ \\
\quad \quad \quad \quad \quad \texttt{(+ \_1 \_2)))))))
\]

If \texttt{web-read/k} cannot behave in a privileged fashion, then its receiver argument will not be invoked automat-
ically by the server. Instead, the entire computation will terminate with the first interaction.

To reflect this problem, let us use a different primitive, \texttt{web-read/r} in place of \texttt{web-read/k}. The suffix
indicates that it will be given the \textit{name} of a receiver as a second argument. \texttt{web-read/r} uses this name
in the URL inserted in the action field of the generated form. To do so, however, each receiver must be a
named Web application that the server can invoke directly, whereas the receivers are currently anonymous
procedures nested within other procedures!

The process of making nested procedures into top-level ones is known as \textit{lifting}. That is, each anony-
mous procedure is moved to the top-level and given an unique name. In the example program above, the
innermost procedure might become

\[
(\texttt{define (f2 \_2)} \ \\
\quad \texttt{(web-display}} \ \\
\quad \quad \texttt{(+ \_1} \ \\
\quad \quad \quad \texttt{\_2))))
\]

which the outer procedure can refer to:

\[
(\texttt{define (f1 \_1)} \ \\
\quad \texttt{(web-read/r "Second number: "} \ \\
\quad \quad \texttt{"f2"}))
\]

The main program then becomes

\[
(\texttt{web-read/r "First number: "} \ \\
\quad \texttt{"f1"})
\]

Note that we intend for \texttt{web-read/r} to be able to terminate safely after dispatching its form. All the remaining
work must be completed by the top-level procedure named in the second argument (in particular, this is now
a string, rather than a procedural value). Each top-level procedure consumes one argument, which is the
data provided by the user.
Unfortunately, by sloppily lifting the procedures to the top-level, we’ve created a problem: $\bullet_1$ is a free identifier in $f2$! The problem is that we were simplistic in the way we lifted the procedures. (A different simplistic method—failing to keep the two instances of $\bullet$ separate—would, of course, have created a different problem, namely that $f2$ would have just added $\bullet$ to itself, ignoring the user’s first input.)

In general, when lifting we must add parameters for all the free variables of the procedure being lifted, then pass along the values for parameters from the point of closure creation. In general, procedure lifting requires the computation of a transitive closure (because lifting one procedure may render another procedure’s previously-bound identifiers free). That is, the Web program ought to become:

\[
\begin{align*}
&\text{(define } (f2 \bullet_1 \bullet_2)) \\
&\quad (\text{web-display} \\
&\quad \quad (+ \bullet_1 \\
&\quad \quad \quad \bullet_2))) \\
&\text{(define } (f1 \bullet_1)) \\
&\quad (\text{web-read/r } "\text{Second number: }" \\
&\quad \quad "f2") \\
&\quad (\text{web-read/r } "\text{First number: }" \\
&\quad \quad "f1")
\end{align*}
\]

But how is $f2$ to get this extra argument? Recall that each top-level procedure takes only one argument: the user’s input. The (traditional) Web server can’t know that it has to hold on to this value and communicate it.

In practice, the Web protocol provides a technique for burying such values in forms using the HTML construct known as a hidden field. Every top-level receiver has to be sensitive to creating and extracting these form values. Specifically, the converted Web application has the following form:

\[
\begin{align*}
&\text{(define } (f2 \text{ user-input}) \\
&\quad (\text{local } [(\text{define } n_1 \quad (\text{get-form-field user-input }'n_1)] \\
&\quad \quad [\text{define } n_2 \quad (\text{get-form-field user-input }'n_2)]) \\
&\quad (\text{web-display} \\
&\quad \quad (+ \bullet_1 \\
&\quad \quad \quad \bullet_2)))) \\
&\text{(define } (f1 \text{ user-input}) \\
&\quad (\text{web-read/r/fields } "\text{Second number: }" \\
&\quad \quad "f2" \\
&\quad \quad \quad \text{user-input} \\
&\quad \quad \quad \quad (\text{list }'n_1)) \\
&\quad (\text{web-read/r } "\text{First number: }" \\
&\quad \quad "f1")
\end{align*}
\]

where $n_1$ and $n_2$ are the names used in the form. The procedure web-read/r/fields takes the same first two inputs as web-read/r. The third argument is the data structure representing the user’s inputs. This is followed by a list of field names; these fields are extracted from the user input and inserted into the generated HTML form using hidden fields.

How would $f1$ know which parameters must be passed to $f2$ using the hidden fields? Because the lifting process would (at least conceptually) generate the following intermediate program:
(web-read/k "First number: "
  ((lambda ()
      (lambda (●₁)
        (web-read/k "Second number: "
          ((lambda (●₁)
              (lambda (●₂)
                (web-display
                 (+ ●₁ ●₂))))
             ●₁))))))

That is, each procedure has been closed over its free variables by the creation of an additional “wrapping” procedure. These otherwise-free variables become the form fields; the inner procedure is the one that is lifted to the top-level.

Exercise 16.4.1 Automate this transformation, i.e., write a program that implements it without the need for human intervention.